Indigo Rose Software

Indigo Rose Plugin SDK

Revision 1.0.1.0

Proprietary Notice
The software described in this document is a proprietary product of Indigo Rose Software Design Corporation and is furnished to the user under a license for use as specified in the license agreement. The software may be used or copied only in accordance with the terms of the agreement.

Information in this document is subject to change without notice and does not represent a commitment on the part of Indigo Rose Software Design Corporation. No part of this document may be reproduced, transmitted, transcribed, stored in any retrieval system, or translated into any language without the express written permission of Indigo Rose Software Design Corporation.

Trademarks
AutoPlay Media Studio, Setup Factory, Visual Patch, TrueUpdate and the Indigo Rose logo are trademarks of Indigo Rose Software Design Corporation. All other trademarks and registered trademarks mentioned in this document are the property of their respective owners.

Copyright
Copyright © 2003-2005 Indigo Rose Software Design Corporation.
All Rights Reserved.

FMOD sound and music system is copyright © Firelight Technologies, Pty Ltd. 1994-2003.
LUA is copyright © 2003 Tecgraf, PUC-Rio.

Table of Contents

INTRODUCTION 8

TOPICS COVERED 8

Buttons *8*

Control Skins *8*

Dependency Plugins *8*

Page Transition Plugins *8*

Action Plugins *8*

Object Plugins *8*

SDK FILES 9

TECHNICAL SUPPORT 9

BUTTONS 10

INTRODUCTION 10

BUTTON FILES EXPLAINED 11

BUTTON STATES 11

Up - Normal *11*

Up - Highlight *11*

Up - Disabled *11*

Down - Normal *12*

Down - Highlight *12*

Down - Disabled *12*

THE MANIFEST FILE 12

<IRButton> *13*

<Version> *14*

<Type> *14*

<Info> *14*

<HitThreshold> *14*

<Caption> *14*

State Sections *15*

THE AUTOPLAY MEDIA STUDIO BUTTON MAKER 15

DISTRIBUTING BUTTON FILES 16

CONTROL SKINS 17

PARAGRAPH SCROLLBARS 17

VIDEO TRANSPORT CONTROLS 19

Distributing Control Skin Files 20

DEPENDENCY PLUGINS 21

Required Skills 21

Detection Files 21

Configuration File 23

Lua Script File 24

Image File 25

Distributing Dependency Plugins 25

PAGE TRANSITION PLUGINS 26

Required Skills 26

Page Transition Files 26

Required Exported Functions 26

irPlg_GetPluginName 26

irPlg_GetPluginVersion 27

irPlg_ShowHelpForPlugin 27

irPlg_GetAuthorInfo 27

irPlg_IsValidLicense 28

irPlg_Transition_GetSettings 28

irPlg_Transition_DoPageTransition 29

irPlg_GetDependencies 30

License Files 30

Distributing Page Transition Plugins 30

Hints and Tips 31

Sample Code 31

Making Transitions Happen 31

Test Your Plugin Thoroughly 31

ACTION PLUGINS 32

Required Skills 32

Action Plugin Files 32

Required Exported Functions 32

irPlg_GetPluginName 33

irPlg_GetPluginVersion 33

irPlg_ShowHelpForPlugin 33

irPlg_ShowHelpForAction 34

irPlg_GetAuthorInfo *34*
irPlg_IsValidLicense *35*
irPlg_GetPluginActionXML *35*
irPlg_GetLuaVersion *35*
irPlg_Action_RegisterActions *36*
irPlg_GetDependencies (OPTIONAL) *36*
SPECIFYING ACTION XML 37
<ActionTemplates> *38*
<Action> *38*
<Name> *38*
<Description> *38*
<ReturnValueType> *38*
<Arguments> *38*
<Arg> *38*
<Name> *39*
<Description> *39*
<Type> *39*
<Default> *39*
<Required> *39*
<EasyMode> *39*
<Default> *39*
<DataType>, <Constraints> *40*
INCLUDING THE LUA LIBRARY 43
Required Files *43*
Adding the Lua Library to Your Project *44*
More About the Lua Library *45*
INCLUDING IRPLUGINHELPERFUNCTIONS 45
DISTRIBUTING ACTION PLUGINS 45
HINTS AND TIPS 45
Sample Code *45*
Test Your Plugin Thoroughly *46*

OBJECT PLUGINS 47
REQUIRED SKILLS 47
OBJECT PLUGIN FILES 47
REQUIRED EXPORTED FUNCTIONS 47
irPlg_GetPluginName *47*

irPlg_GetPluginVersion *47*
irPlg_GetPluginActionXML *48*
irPlg_ShowHelpForAction *48*
irPlg_ShowHelpForPlugin *48*
irPlg_IsValidLicense *48*
irPlg_GetLuaVersion *48*
irPlg_GetAuthorInfo *48*
irPlg_Object_CreateObject *48*
irPlg_Object_DeleteObject *48*
irPlg_GetIRPluginObjectVersion *49*
irPlg_GetDependencies (OPTIONAL) *49*
irPlg_Object_GetFonts (OPTIONAL) *49*
irPlg_Object_TranslateMessage (OPTIONAL) *50*
THE CIRPLUGINOBJECT CLASS 51
CIRPluginObject::GetDefaultSize *51*
CIRPluginObject::IsWindowedObject *51*
CIRPluginObject:: GetWindowHandle *52*
CIRPluginObject::DrawDesign *52*
CIRPluginObject::DrawRun time *53*
CIRPluginObject::GetCustomProperties *53*
CIRPluginObject::SetCustomProperties *54*
CIRPluginObject::ShowProperties *54*
CIRPluginObject::GetNumEvents *54*
CIRPluginObject::GetEvent *55*
CIRPluginObject::RegisterLUAFunctions *55*
CIRPluginObject::LetAMSHandleCursorChange *55*
CIRPluginObject::LetAMSHandleSounds *56*
CIRPluginObject::LetAMSHandleTooltip *56*
CIRPluginObject::CanSetFocus *57*
CIRPluginObject::DoSetFocus *57*
CIRPluginObject::OnMouseOver *57*
CIRPluginObject::OnMouseLeave *58*
CIRPluginObject::OnLBtnDown *58*
CIRPluginObject::OnLBtnUp *58*
CIRPluginObject::OnLBtnDoubleClick *59*
CIRPluginObject::OnRBtnDown *59*
CIRPluginObject::OnRBtnUp *60*

CIRPluginObject::OnRBtnDoubleClick *60*
CIRPluginObject::FireEvent *61*
CIRPluginObject::GetObjectID *61*
CIRPluginObject::ShowWindow *61*
CIRPluginObject::m_pLuaState *62*
CIRPluginObject::m_szObjectID *62*
IRLUA_PLUGIN_GetObjectPtr *62*
IRLUA_PLUGIN_RedrawObject *62*
DISTRIBUTING OBJECT PLUGINS 63
HINTS AND TIPS 63
Sample Code *63*
Test Your Plugin Thoroughly *63*

Introduction

This document covers creating add-ons and plugins for Indigo Rose's suite of development tools. Namely, AutoPlay Media Studio (v5.0+), Setup Factory (v7.0+), TrueUpdate (v2.0+) and Visual Patch (v2.0+). Note that all of the sections in this document do not apply to all of these products. Each section denotes which products the information applies to.

Topics Covered

In this document we will cover the following ways in which to extend Indigo Rose products:

Buttons

Indigo Rose Corporation has created a simple file format for multi-state button files that is used by AutoPlay Media Studio. These buttons are easy to create and there is even a custom-made tool available for their creation and editing.

Control Skins

The paragraph and video objects support the use of custom image maps to represent their scrollbars and transport controls. This allows you to create attractive "skins" for these controls.

Dependency Plugins

Dependency plugins allow you to create scripts that can be used to detect various applications and technologies at run time.

Page Transition Plugins

In this product, the transitions that can occur between pages are implemented through page transition plugins.

Action Plugins

Sometimes you need functionality that is not provided by the product's actions nor possible to create with scripts alone. In these cases, it is possible to create action plugins that can extend the product to do almost anything.

Object Plugins

Object plugins allow you to add new, custom objects to AutoPlay Media Studio. This enables you to create very powerful, flexible applications.

SDK Files

There are a number of files that are provided to help you use this SDK. These files are located in the folder where you installed the SDK. There is a subfolder called “Samples” that contains sample Visual C++ projects (made with Visual C++ 6.0 SP5) as well as some image and button samples and a subfolder called “Includes” that contains the source files needed to create action and object plugins.

Technical Support

Direct email or telephone technical support is not provided by Indigo Rose Corporation for any SDK issues including (but not limited to) creating buttons, plugins or images. However, there is a forum devoted to this subject at http://www.indigorose.com where you can discuss plugin development issues with other developers and Indigo Rose staff.

Buttons

This section applies to:

- AutoPlay Media Studio 5.0
- AutoPlay Media Studio 6.0

Introduction

Button objects are powerful objects that are natively supported in AutoPlay Media Studio. Buttons are simply interactive graphical elements that support multiple states as well as caption text.

Here is a button in normal state:

Here it is when the mouse cursor passes over it:

There are actually two types of buttons, or more correctly, two ways in which a button can be used: as a push button, or a toggle button.

A push button (called "Standard" in AutoPlay Media Studio) is the most common type of button and is the kind you usually see in Windows programs. A push button "presses" down when the mouse clicks on it and then comes back up when the mouse button is released or the mouse cursor leaves the button's area.

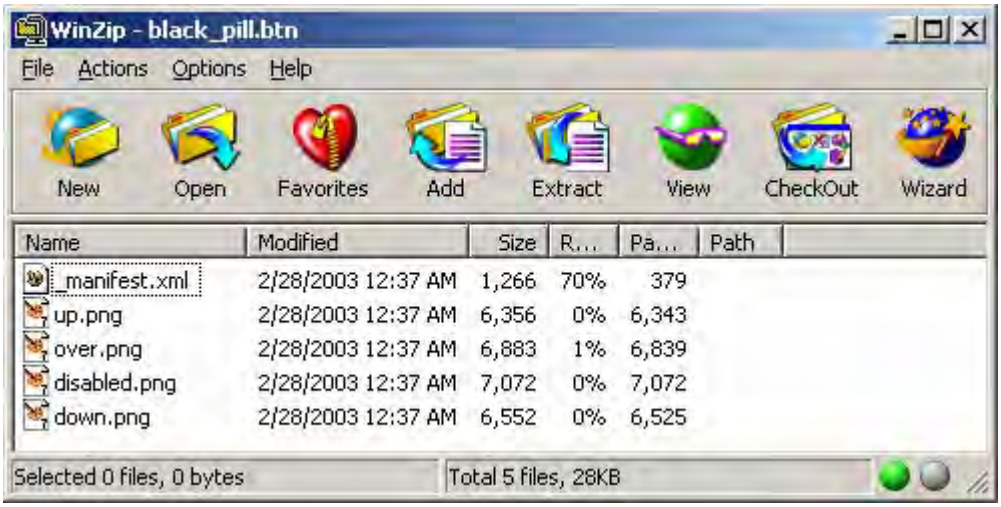
A toggle button is a button that remembers its state. That is, the first time that you click it with the mouse, it stays down. The next time you click it, it comes back up. Either way, it remains in the last state that you put it in, either up or down.

Notice how there is actually different images displayed for each button state, even though a button is just one file (in the case above "black_pill.btn".)

Note: The “black_pill.btn” file is in the \Samples subfolder of the SDK folder.

Button Files Explained

Button files have the file extension ".btn". These files are simply zip files with certain image and data files inside of them. You can see this by opening a button file with WinZip or a similar zip utility.


WinZip - black_pill.btn
File Actions Options Help
New Open Favorites Add Extract View CheckOut Wizard
Name Modified Size R... Pa... Path
_manifest.xml 2/28/2003 12:37 AM 1,266 70% 379
up.png 2/28/2003 12:37 AM 6,356 0% 6,343
over.png 2/28/2003 12:37 AM 6,883 1% 6,839
disabled.png 2/28/2003 12:37 AM 7,072 0% 7,072
down.png 2/28/2003 12:37 AM 6,552 0% 6,525
Selected 0 files, 0 bytes
Total 5 files, 28KB

The button file is made up of one to six images as well as a file called "_manifest.xml". Each image file can represent one or more of the six supported button states.

Button States

The six states supported by button files are:

Up - Normal

This is the state that a push button is normally in. This is the default state that will be displayed if the mouse is not over and/or clicking the button and the button is not disabled.

In the case of a toggle button, this is the default view of the button when it is in the "up" state.

Up - Highlight

This state is displayed when the mouse pointer is over the button and it is in the up state. This state usually provides the user with some sort of visual feedback that the mouse is over the button.

Up - Disabled

This state is displayed if the button is in the up state and is disabled. Whether or not a button is disabled (or even can be disabled) is up to the program that uses the button. Usually this state is a "grayed-out" version of the button that will signify to the user that it is not interactive.

Down - Normal

This state is displayed if the user has the mouse over a push button with the left mouse button down or a toggle button is in the down state but does not have the mouse over it.

Down - Highlight

This state is displayed when the mouse pointer is over the button and it is in the down state. This state usually provides the user with some sort of visual feedback that the mouse is over the button.

Down - Disabled

This state is displayed if the button is in the down state and is disabled. Whether or not a button is disabled (or even can be disabled) is up to the program that uses the button. Usually this state is a "grayed-out" version of the button that will signify to the user that it is not interactive.

Note that this state will only be displayed if the button is a toggle button because a push button will only show the Up - Disabled state.

The Manifest File

Every button contains a manifest file called "_manifest.xml". This file contains all of the information that AutoPlay Media Studio or any other program needs to know in order to properly display a button file. The manifest file is in XML format. XML is simply specially marked text that allows you to organize and describe data in a text file. You can open an XML file with notepad or any other text editor.

Try extracting and opening the manifest file of a button file. You will see something like this:

```
<IR_Button>
    <Version>1.0</Version>
    <Type>0</Type>
    <Info>
      <Name>Blue Diamond</Name>
      <Desc>Shimmering water upon a deep blue background</Desc>
      <Author>Indigo Rose Software</Author>
      <Copyright>Copyright © 2003 Indigo Rose Software.</Copyright>
      <URL>http://www.indigorose.com</URL>
      <Email>sales@indigorose.com</Email>
      <Other></Other>
    </Info>
    <HitThreshold>200</HitThreshold>
    <Caption>
      <Text>Click Here</Text>
      <Font>
            <Face>Arial</Face>
            <CharSet>0</CharSet>
            <Size>12</Size>
            <Bold>0</Bold>
            <Italic>0</<Italic>
```

```
            <Underline>0</Underline>
            <Strikeout>0</Strikeout>
      </Font>
      <CenterX>50</CenterX>
      <CenterY>50</CenterY>
    </Caption>
    <Up>
      <Normal>
            <Image>up.png</Image>
            <FontColor>FFFFFF</FontColor>
            <DeltaX>0</DeltaX>
            <DeltaY>0</DeltaY>
      </Normal>
      <Highlight>
            <Image>up_high.png</Image>
            <FontColor>FFFFFF</FontColor>
            <DeltaX>0</DeltaX>
            <DeltaY>0</DeltaY>
      </Highlight>
      <Disabled>
            <Image>up_dis.png</Image>
            <FontColor>FFFFFF</FontColor>
            <DeltaX>0</DeltaX>
            <DeltaY>0</DeltaY>
      </Disabled>
    </Up>
    <Down>
      <Normal>
            <Image>down.png</Image>
            <FontColor>FFFFFF</FontColor>
            <DeltaX>5</DeltaX>
            <DeltaY>0</DeltaY>
      </Normal>
      <Highlight>
            <Image>down_high.png</Image>
            <FontColor>FFFFFF</FontColor>
            <DeltaX>5</DeltaX>
            <DeltaY>0</DeltaY>
      </Highlight>
      <Disabled>
            <Image>down_dis.png</Image>
            <FontColor>FFFFFF</FontColor>
            <DeltaX>5</DeltaX>
            <DeltaY>0</DeltaY>
      </Disabled>
    </Down>
</IR_Button>
```

What follows in this section is a breakdown of the specific sections of the manifest file.

<IRButton>

This is the main XML tag of the button file. It must surround all other button data. Without this tag, the button file will not be recognized as valid.

<Version>

This identifies the version of the button file. As of this writing, all button files whould use "1.0" as the version. This field is there so that if the button file format changes in the future, programs can identify which version of the file format it uses.

<Type>

The default type of the button; 0 = push button, 1 = toggle button. Note that this just tells the program that uses the button what the preferred or default button type is and does not necessarily mean that the button will not or cannot be used with the other style.

<Info>

This section contains a number of fields that can be used to specify custom descriptive information about the button and its author. Although all of these fields are optional, it is a good idea to at least fill in the <Name> and <Desc> fields.

<HitThreshold>

The alpha value at or below which a mouse hit will return FALSE. Use –1 to have the mouse always hit at all points. This allows you to create the effect of non-rectangular objects.

You see, button files support 32-bit images with alpha channels (in the form of PNG files). By setting this threshold, you can tell the host program that the button is not "hit" if the mouse is over a transparent or semi-transparent area of the current state's image. If you do not understand this feature, it is usually safe to leave the setting at 200.

<Caption>

This section lets you specify defaults for the caption text of the button file. Note that it is up to the host program whether to use these defaults or to show a caption at all. In AutoPlay Media Studio 5.0, these caption defaults are used as the default caption settings for the object.

Here is a brief description of the sub-fields of the <Caption> section:

Field Name	**Description**
Text	The default text for the button.
Font > Face	The face name of the font.
Font > CharSet	A numerical value for the character set. Use 0 for ASCI_CHARSET (Default)
Font > Size	The size of the font in pixels.
Font > Bold	Whether the font is bold by default. 0 = No, 1 = Yes

Font > Italic	Whether the font is italicized by default. 0 = No, 1 = Yes
Font > Underlined	Whether the font is underlined by default. 0 = No, 1 = Yes
Font > Strikeout	Whether the font is striked out by default. 0 = No, 1 = Yes
CenterX	The horizontal center of the button (in percent) for text placement purposes. Use 50 to center the caption horizontally.
CenterY	The vertical center of the button (in percent) for text placement purposes. Use 50 to center the caption vertically.

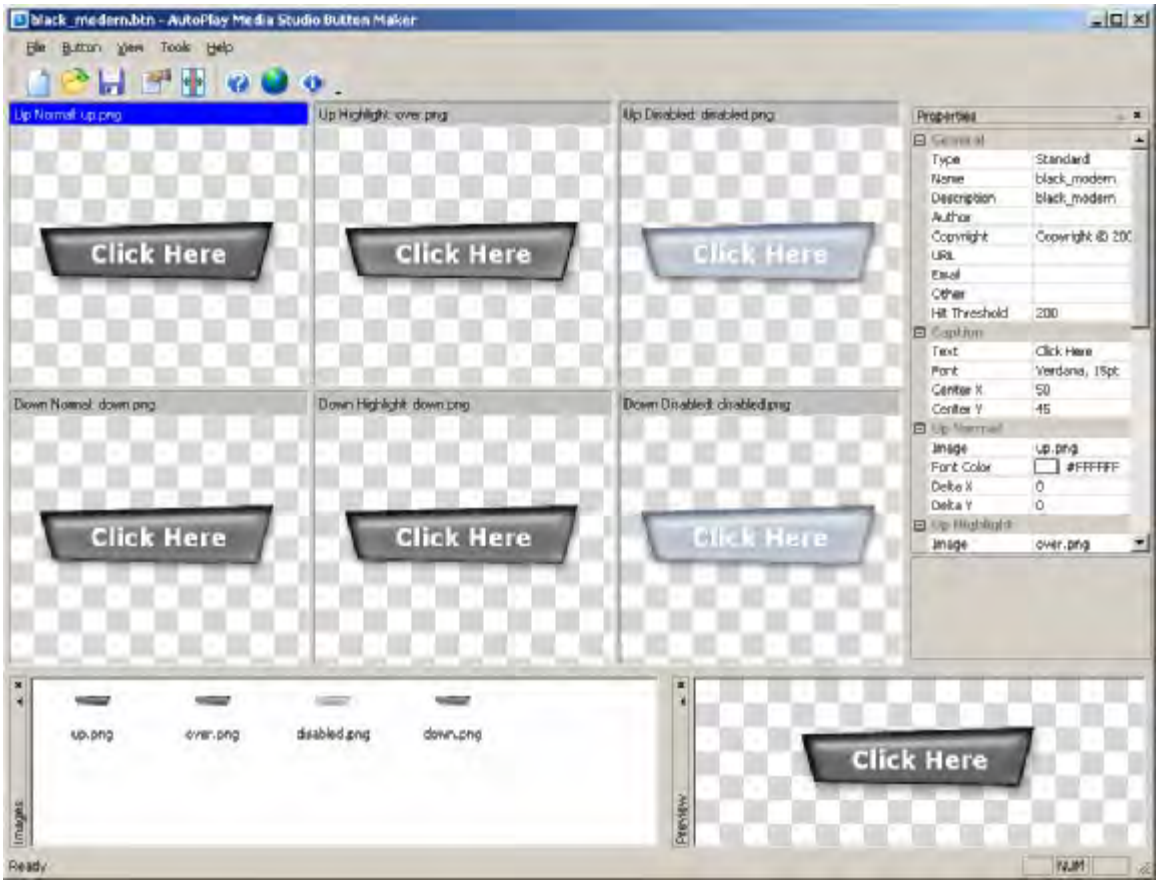
State Sections

The rest of the sections represent options for the specific states of the button. There are two major sections, "<Up>" and "<Down>". Both of these major sections each have three sub-sections called "<Normal>", "<Highlight>" and "<Disabled>". Here is a description of each of the states' sub-fields:

Field Name	**Description**
Image	The name of the image file to use for the state. It should just be the name of the file without a path such as "up_normal.png". You can use bmp, jpg, pcx, png, tga, tif (uncompressed), pcx, wmf, apm, emf, psd or pcd image files. However, we recommend using png files. Note that you can use the same image for more than one state and that every state MUST specify an image name.
Font Color	A 24-bit hexadecimal color value (e.g. FF00FF) that represents the default caption color of the text for the state.
DeltaX	The amount (in percent) to adjust the button’s horizontal center for each state. Can be a negative value.
DeltaY	The amount (in percent) to adjust the button’s vertical center for each state. Can be a negative value.

The AutoPlay Media Studio Button Maker

Indigo Rose Corporation has created a separate WYSIWYG environment for creating button files in a more natural, visual manner rather than having to assemble the button file manually. This editor is shipped along with AutoPlay Media Studio. You can access it be selecting Tools > Button Maker from the menu.



Distributing Button Files

Button files can be dragged and dropped onto AutoPlay Media Studio from any folder on your system from Windows Explorer. However, if you would like to make your buttons readily available to your projects, they should be copied to the Gallery folder.

The Gallery folder is a subfolder of the AutoPlay Media Studio installation folder. Usually, it is located at:

"C:\Program Files\AutoPlay Media Studio (version#)\Gallery"

Specifically, you should copy your button files to a custom subfolder of the "Gallery\Buttons" folder. For example:

"C:\Program Files\AutoPlay Media Studio (version#)\Gallery\Buttons\My_Buttons"

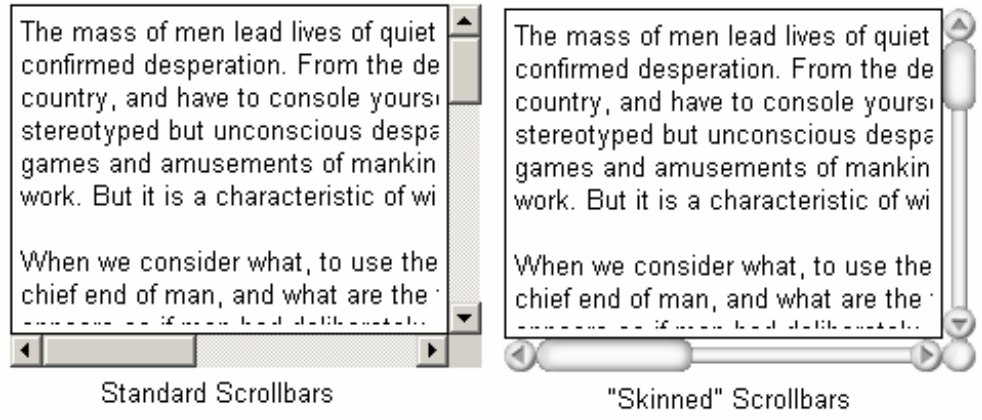
This way they will show up in the Gallery pane inside of AutoPlay Media Studio and will be easily accessible.

Control Skins

This section applies to:

- AutoPlay Media Studio 5.0
- AutoPlay Media Studio 6.0

There are two built-in AutoPlay Media Studio objects that support custom "skinning". They are the paragraph and video objects. Both of these objects have default views that don't use skins, but they can both be enabled to use custom skins as well.


The mass of men lead lives of quiet
confirmed desperation. From the de
country, and have to console yours
stereotyped but unconscious despa
games and amusements of mankin
work. But it is a characteristic of wi
When we consider what, to use the
chief end of man, and what are the
Standard Scrollbars
"Skinned" Scrollbars

These skin files are nothing more than image files with certain specific requirements. The sections below will describe these image files in more detail.

Paragraph Scrollbars

Paragraph scrollbars are really one large image made up of 19 smaller images. Each small image must be square (height and width the same) and can vary in size from 16x16 to 48x48. So, for example, a paragraph scrollbar image that contains 16x16 images must be 304x16 pixels in size. For example:

Note: The above image is called “sb_Corporate.png” and is included in the \Samples subfolder in the SDK folder.

These images can be 24-bit (non-transparent) or 32-bit (with alpha transparency - png only). Here is a description of each image indexed from 1 at the left to 19 at the right:

Index	Use
1	The up arrow in normal state. This is the arrow that points upward located at the top of the vertical scrollbar. Pressing this arrow scrolls the text up.
2	The up arrow when the left mouse button is pressed on it. This is the arrow that points upward located at the top of the vertical scrollbar. Pressing this arrow scrolls the text up.
3	The down arrow in normal state. This is the arrow that points downward located at the bottom of the vertical scrollbar. Pressing this arrow scrolls the text down.
4	The down arrow when the left mouse button is pressed on it. This is the arrow that points downward located at the bottom of the vertical scrollbar. Pressing this arrow scrolls the text down.
5	The left arrow in normal state. This is the arrow that points leftward located at the left of the horizontal scrollbar. Pressing this arrow scrolls the text left.
6	The left arrow when the left mouse button is pressed on it. This is the arrow that points leftward located at the left of the horizontal scrollbar. Pressing this arrow scrolls the text left.
7	The right arrow in normal state. This is the arrow that points rightward located at the right of the horizontal scrollbar. Pressing this arrow scrolls the text right.
8	The right arrow when the right mouse button is pressed on it. This is the arrow that points rightward located at the right of the horizontal scrollbar. Pressing this arrow scrolls the text right.
9	The top portion of the vertical thumbtack. This will be used if the size of the vertical thumbtack is large enough to accommodate a multi-part thumbtack. Otherwise image 12 will be used.
10	The middle portion of the vertical thumbtack. This will be used if the size of the vertical thumbtack is large enough to accommodate a multi-part thumbtack. Otherwise image 12 will be used.
11	The bottom portion of the vertical thumbtack. This will be used if the size of the vertical thumbtack is large enough to accommodate a multi-

	part thumbtack. Otherwise image 12 will be used.
12	The small vertical thumbtack. This will be used if the size of the vertical thumbtack is too small to accommodate a multi-part thumbtack. Otherwise images 9-11 will be used.
13	The left portion of the horizontal thumbtack. This will be used if the size of the horizontal thumbtack is large enough to accommodate a multi-part thumbtack. Otherwise image 16 will be used.
14	The center portion of the horizontal thumbtack. This will be used if the size of the horizontal thumbtack is large enough to accommodate a multi-part thumbtack. Otherwise image 16 will be used.
15	The right portion of the horizontal thumbtack. This will be used if the size of the horizontal thumbtack is large enough to accommodate a multi-part thumbtack. Otherwise image 16 will be used.
16	The small horizontal thumbtack. This will be used if the size of the horizontal thumbtack is too small to accommodate a multi-part thumbtack. Otherwise images 13-15 will be used.
17	The tracking area behind the thumbtack on the vertical scrollbar.
18	The tracking area behind the thumbtack on the horizontal scrollbar.
19	The area in the bottom-right corner of the paragraph object if both vertical and horizontal scrollbars are displayed.

Video Transport Controls

The custom video transport controls are really one large image made up of 15 smaller images. Each small image must be square (height and width the same) and can vary in size from 16x16 to 48x48. So, for example, a paragraph scrollbar image that contains 16x16 images must be 240x16 pixels in size. For example:

Note: The above image is called “vt_Corporate.png” file is in the \Samples subfolder of the SDK folder.

These images can be 24-bit (non-transparent) or 32-bit (with alpha transparency - png only). Here is a description of each image indexed from 1 at the left to 15 at the right:

Index	**Use**
1	The play button in normal state.
2	The play button when the mouse is over it but the left mouse button is

	not pressed.
3	The play button when the mouse is over it and the left mouse button is pressed.
4	The pause button in normal state.
5	The pause button when the mouse is over it but the left mouse button is not pressed.
6	The pause button when the mouse is over it and the left mouse button is pressed.
7	The stop button in normal state.
8	The stop button when the mouse is over it but the left mouse button is not pressed.
9	The stop button when the mouse is over it and the left mouse button is pressed.
10	The left portion of the slider area behind the slider thumbtack.
11	The center portion of the slider area behind the slider thumbtack.
12	The right portion of the slider area behind the slider thumbtack.
13	The slider thumbtack button in normal state.
14	The slider thumbtack button when the mouse is over it but the left mouse button is not pressed.
15	The slider thumbtack button when the mouse is over it and the left mouse button is pressed.

Distributing Control Skin Files

Control skin files are automatically scanned and ready for use in AutoPlay Media Studio as long as they are in the correct folders.

Scrollbar skins should be located in the \Plugins\Scrollbars subfolder of the AutoPlay Media Studio installation folder. In order to be recognized, all scrollbar skins must start with the prefix “sb_”. For example, “sb_Funky Red.png”.

Video transport skins should be located in the \Plugins\Transports subfolder of the AutoPlay Media Studio installation folder. In order to be recognized, all scrollbar skins must start with the prefix “vt_”. For example, “vt_Sunset.png”.

Dependency Plugins

This section applies to:

- AutoPlay Media Studio 5.0
- AutoPlay Media Studio 6.0

Dependency plugins are files that tell AutoPlay Media Studio how to detect certain technologies and applications at run time.

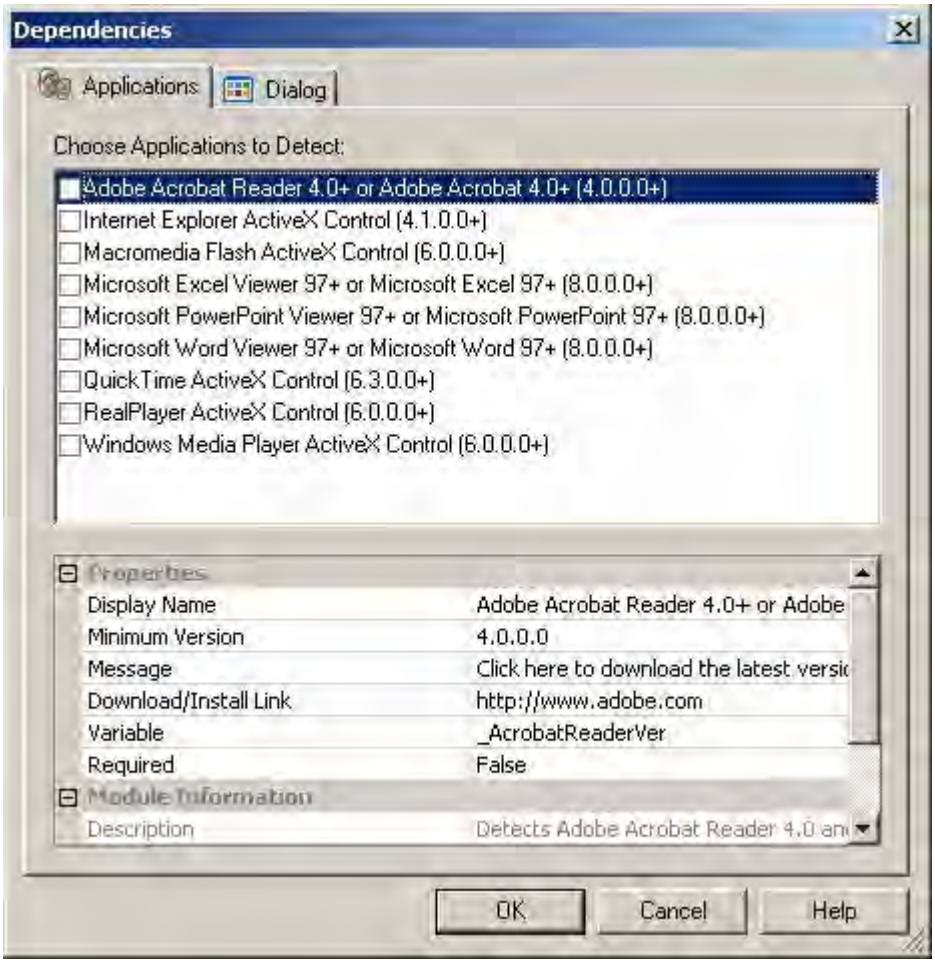
Required Skills

In order to create dependency plugins you will need to be proficient with AutoPlay Media Studio's scripting language and the available actions. Other minor skills include the ability to create and edit zip archives as well as a familiarity with XML.

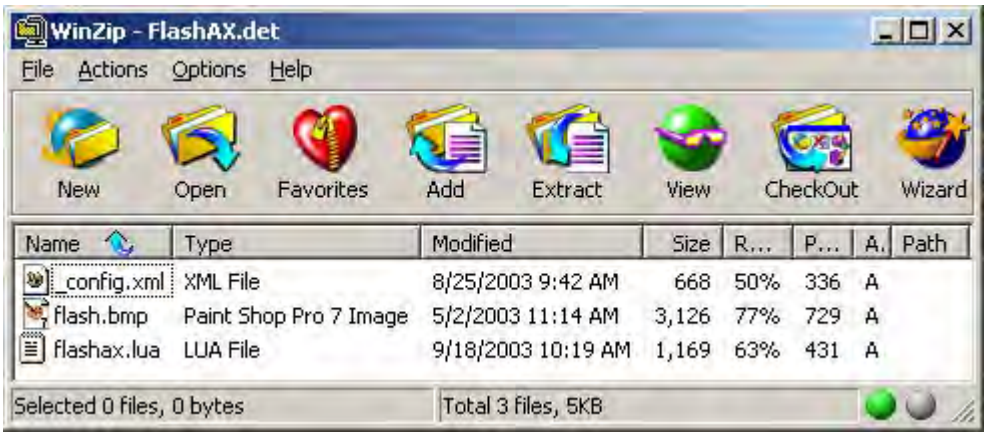
Detection Files

Dependency plugins are implemented in files called detection files that have the file extension ".det". They are located in the \Plugins\Detect subfolder of the AutoPlay Media Studio installation folder.

Detection files are available to AutoPlay Media Studio projects through the Dependencies screen (select Project > Dependencies from the menu.) This screen dynamically displays all available detection files.



Detection files are actually zip archives with several files within them. You can open and view the contents of a detection file by opening it with WinZip or a similar zip program.



Note: There is a detection file called “FlashAX.det” file is in the \Samples subfolder of the SDK folder.

Detection files contain three components; a configuration file, a Lua script file, and an image file.

Configuration File

The configuration file is an XML file and can be edited with any text editor. Each detecion file must contain a configuration file called "_config.xml". Here is a sample configuration file:

```
<MissingTechConfig>
    <Info>
      <Description>Detects Macromedia Flash ActiveX
Control</Description>
      <Author>Indigo Rose Corporation</Author>
      <Email>support@indigorose.com</Email>
      <Web>http://www.indigorose.com</Web>
      <Copyright>Copyright © 2003 Indigo Rose Corporation</Copyright>
    </Info>
    <Name>Macromedia Flash ActiveX Control</Name>
    <ScriptFile>flashax.lua</ScriptFile>
    <ScriptFcn>ir_GetFlashAXVersion</ScriptFcn>
    <Image>flash.bmp</Image>
    <DefMinVer>6.0.0.0</DefMinVer>
    <DefMessage>Click here to download the newest Flash
control.</DefMessage>
    <DefLink>http://www.macromedia.com</DefLink>
    <DefVar>_FlashVer</DefVar>
</MissingTechConfig>
```

<Info>

Allows you to specify information about you, the plugin author. This information will be displayed on the Dependencies screen at design time.

<Name>

The name of the dependency that you are detecting. This name should precisely describe the technology or application that your script will detect. For example, if you make a detection file that detects Microsoft Word, make sure that you specify in the name the exact version that it will detect. For example, if you have tested and verified that the script detects Microsoft Word 97 and up, but have not tested it with earlier versions, specify that in the name by using something like "Microsoft Word 97".

<ScriptFile>

The name of your Lua script file within the detection (zip) file.

<ScriptFnc>

The name of the function within the ScriptFile that should be called in order to detect the technology. More details about the script file and script function are in the next section.

<Image>

The name of the image file that will be used to represent the technology at run time. The file named here must exist in the detection (zip) file.

<DefMinVer>

The default minimum version of the technology or application that should be used for the dependency plugin at design time. The user can change this value according to their needs. Try to choose a version that you think would be the most commonly needed by developers.

<DefMessage>

The default instructions that will appear with the missing technology at run time. The user can change this according to their needs at design time.

<DefLink>

The default link that will be executed if the end user double-clicks on the technology at run time. The user can change this according to their needs at design time. Although this can be a link to a local file, it is best to use a Web address as a default.

<DefVar>

The default variable that will be used to store the detected version of the technology at run time. The user can change this according to their needs at design time. Try to use a variable name that is very specific to your dependency plugin. It is also a good idea to begin the variable name with an underscore to help avoid variable naming conflicts.

Lua Script File

Every dependency plugin must contain a Lua script file that actually does the work of detecting the technology or application. These scripts can use any and all available AutoPlay Media Studio actions as well as the standard Lua language.

Here is the abbreviated content of a sample script file:

```
function ir_GetFlashAXVersion()
    strVersion = “0.0.0.0”;

    -- Do your detection here are

    return strVersion;
end
```

Here are the important things to know about your script file:

1. It must have a function that takes no arguments and always returns a string that contains a version number (even if the version number is “0.0.0.0”).

2. Version numbers should be in the #.#.#.# format whenever possible.

3. The script file can contain other variables and functions, but only one function can be specified as the one that gets called at run time. And again, that one function must conform to the guidelines in (1).

4. The function can have any valid Lua function name. However, because it will be loaded into the global Lua engine at run time, it should be a unique name. Consider prefixing it with your initials or some other identifier. Don't, for example, use "`function Detect()`".

5. Your script file should be well coded, documented and tested. Don't leave room for a run time error.

Image File

The image file is a small bmp image that will be used to represent your technology at run time.

Missing Applications
This program requires the following free technologies which cannot be located on your system.
Please double-click on an item to find out how to install the technology.
Macromedia Flash ActiveX Control
6.0.0.0
http://www.macromedia.com
Click here to download the newest Flash control.
Abort
Continue

The image file should be a 32x32 bitmap. The color depth should be 8 or 24 bits-per-pixel. Make any parts of the image that you want to be transparent RGB 255, 0, 255 (#FF00FF).

Distributing Dependency Plugins

Dependency plugins (detection files) should be copied to the \Plugins\Detect subfolder of the AutoPlay Media Studio installation folder.

Page Transition Plugins

This section applies to:

- AutoPlay Media Studio 5.0
- AutoPlay Media Studio 6.0

Page transition plugins allow you to create different page transition effects at run time.

Required Skills

You will need the following skills to create page transition plugins:

- C or C++ programming
- Be able to create Windows DLLs
- Be familiar with the Windows GDI API

Page Transition Files

Page transition files are Windows DLLs that have the file extension ".tns". These DLLs expose a certain set of exported functions that AutoPlay Media Studio uses for informational and functional purposes.

Required Exported Functions

Below is a list and description of functions that must be exported from your page transition plugin DLL.

irPlg_GetPluginName

Purpose: To return the name of the transition effect to the calling program.

Prototype:

```
int irPlg_GetPluginName(char* szBuffer, int* pnBufferSize)
```

Parameters:

szBuffer - [out] A pointer to a character buffer that will receive the name of the transition.
pnBufferSize - [in/out] A pointer to an integer that contains the number of characters in szBuffer on the way in and will be set to the number of characters actually copied to the buffer on the way out.

Returns:

The number of characters copied to the buffer or -1 if the buffer was not large enough to contain the transition's name. If you return -1, be sure that you set pnBufferSize to the number of characters actually required.

irPlg_GetPluginVersion

Purpose: To return the version of the transition effect to the calling program. This version number is used to identify different versions of your plugin. It can be any version number, but it should be in the format "#.#.#.#".

Prototype:

```
int irPlg_GetPluginVersion (char* szBuffer, int* pnBufferSize)
```

Parameters:

szBuffer - [out] A pointer to a character buffer that will receive the version of the transition.
pnBufferSize - [in/out] A pointer to an integer that contains the number of characters in szBuffer on the way in and will be set to the number of characters actually copied to the buffer on the way out.

Returns:

The number of characters copied to the buffer or -1 if the buffer was not large enough to contain the transition's version. If you return -1, be sure that you set pnBufferSize to the number of characters actually required.

irPlg_ShowHelpForPlugin

Purpose: To show help information for the transition plugin.

Note: This function is not currently called from AutoPlay Media Studio 5.0. It is here for compatibility purposes and for possible future use. For now it is fine to just return TRUE.

Prototype:

```
bool irPlg_ShowHelpForPlugin(char* lpszPluginPath, HWND hParentWnd)
```

Parameters:

lpszPluginPath - [in] A pointer to a character buffer that contains the folder that the plugin is currently located in. This can be useful if you want to open a help file from the same folder.
hParentWnd - [in] A handle to the window that is calling the function. It is sometimes necessary to have this value when opening files.

Returns: TRUE if the help was successfully displayed or FALSE if it failed.

irPlg_GetAuthorInfo

Purpose: To return information about the plugin and its author. This information will be displayed in the About Plugin screen at design time.

Prototype:

```
int irPlg_GetAuthorInfo (char* szBuffer, int* pnBufferSize)
```

Parameters:

szBuffer - [out] A pointer to a character buffer that will receive the author info of the transition. This information can contain and be formatted in any way that you wish
pnBufferSize - [in/out] A pointer to an integer that contains the number of characters in szBuffer on the way in and will be set to the number of characters actually copied to the buffer on the way out.

Returns:

The number of characters copied to the buffer or -1 if the buffer was not large enough to contain the transition's author information. If you return -1, be sure that you set pnBufferSize to the number of characters actually required.

irPlg_IsValidLicense

Purpose: To determine if license information for the plugin is valid. The license information comes from a license file that is located in the same folder as your plugin DLL at design time. License files are covered in more detail in the topic "License Files" on page 30.

Prototype:

```
bool irPlg_IsValidLicense (char* lpszLicenseInfo)
```

Parameters:

lpszLicenseInfo - [in] A pointer to a character buffer that contains text from the license file.

Returns:

TRUE is the license information is valid or FALSE if it is not. If you return FALSE, the user will not be able to use the transition.

irPlg_Transition_GetSettings

Purpose: To display and update the transition's custom settings. This function takes in the current settings of the plugin, displays a dialog that allows the user to change the settings and then returns the new settings back to the calling program. Note that the transition's settings are completely arbitrary and specific to the transition. You can supply any kind and amount of data as long as it is valid ASCII text.

Prototype:

```
int irPlg_Transition_GetSettings(char* szCurrentSettings,
      HWND hParent, char* szNewSettings, int* nNewSettingsSize)
```

Parameters:

szCurrentSettings - [in] A pointer to a character buffer that contains the current settings. This string can be empty if it was not previously initialized.

hParent - [in] A handle to the parent window. This can be useful when showing properties dialogs as child windows.
szNewSettings - [out] A pointer to a character buffer that will receive the new settings for the transition.
nNewSettingsSize - [in/out] A pointer to an integer that contains the number of characters in szNewSettings on the way in and will be set to the number of characters actually copied to the buffer on the way out.

Returns:

The number of characters copied to the buffer or -1 if the buffer was not large enough to contain the transition's settings. If you return -1, be sure that you set pnBufferSize to the number of characters actually required.

irPlg_Transition_DoPageTransition

Purpose: This is the function that actually gets called at run time to perform the transition on the screen.

Prototype:

```
void irPlg_Transition_DoPageTransition(HDC hCurrentPage, HDC hNextPage,
      SIZE sizeImage, POINT ptOffset, HWND hWindow, char* szSettings)
```

Parameters:

hCurrentPage - [in] A device context handle to the current page (that is, the page that is being transitioned from.) This DC is the size of the entire client area of the run time's window.
hNextPage - [in] A device context handle to the next page (that is, the page that is being transitioned to.) This DC is the size of the entire client area of the run time's window.
sizeImage - [in] The size the area that the transition should be applied to. Note that this is usually, but not necessarily, the size of the client area of the run time's window. In the case of a run time that is in kiosk mode, this area may be smaller than the actual DC because of the way that the page is drawn in the center of the client area.
ptOffset - [in] A POINT structure that contains the x and y offsets of the page area (the area that the transition should be applied to) relative to the upper-left corner of the run time window's client area. In most cases this is x = 0, y = 0 except in the case of a run time in kiosk mode in which case the page area may be centered in the client area. Either way, your transition should always take this point into consideration.
hWindow - [in] A handle to the run time view's window. Normally this is not needed.
szSettings - [in] A pointer to a character buffer that contains the current settings. How you interpret this information is specific to your plugin.

Returns:

Nothing.

irPlg_GetDependencies

Purpose: To return additional dependency files required by the plugin. This function should return only the filenames of files needed by the plugin at run time in a bar ("|") separated string. The files must be located in the same folder as the plugin file at design time. For example, "support.dll|splash.png", tells AutoPlay Media Studio to collect and bring along the files support.dll and splash.png with the plugin at build time. These files will be copied to the same folder as the plugin file.

Prototype:

```
int irPlg_GetDependencies (char* szBuffer, int* pnBufferSize)
```

Parameters:

szBuffer - [out] A pointer to a character buffer that will receive the list of dependency files.
pnBufferSize - [in/out] A pointer to an integer that contains the number of characters in szBuffer on the way in and will be set to the number of characters actually copied to the buffer on the way out.

Returns:

The number of characters copied to the buffer or -1 if the buffer was not large enough to contain the transition's name. If you return -1, be sure that you set pnBufferSize to the number of characters actually required.

License Files

Page transition, action and object plugins use license files in order to protect the usage and distribution of the plugin. License files are simply text files with the file extension ".lic". These files can be empty or contain any text that you want, but the file must exist in order to be used at design time. License files should have the same name as the plugin's DLL file with the ".lic" extension. So, for example, if your plugin is called "SuperEffect.tns", your license file must be called "SuperEffect.lic" and be located in the same folder at design time.

License files are not distributed or used at run time. At design time AutoPlay Media Studio reads the contents of the license file and then calls the irPlg_IsValidLicense function to determine if the license is valid. If the license file is not found or the irPlg_IsValidLicense function returns FALSE, the plugin will not be available to the developer.

Distributing Page Transition Plugins

The page transition plugin DLL (".tns" file) and the license file (".lic") should be copied to the \Plugins\Transitions subfolder of the AutoPlay Media Studio installation folder in order to be visible to the design environment.

Hints and Tips

Here are some things that may help you out when creating page transition plugins.

Sample Code

An entire sample Visual C++ 6.0 project is available for you to see and learn from. It is the source code used to produce the Wipe transition that ships with AutoPlay Media Studio. This project is located in the \Samples\IRWipeTransition subfolder of the SDK installation folder.

This project will show you all of the details behind making your own page transition plugin. Note that this DLL statically links to MFC for the sake of the properties dialog. However, it is possible to make page transition plugins that don't rely on MFC.

Making Transitions Happen

The best way to make a transition happen is to use the Windows API function BitBlt to copy rectangle areas from the hNextPage to hCurrentPage. Take a look at the Wipe transition source code to see how it is done. Of course, you can use any method that you wish assuming that you can do it with the variable passed into the irPlg_Transition_DoPageTransition function.

Test Your Plugin Thoroughly

Make sure to fully test your plugin before sharing it with others. Try it with all sorts of window sizes and types. Make sure that you test it with the kiosk mode. It is important to test it thoroughly because a buggy plugin could cause the entire run time to crash.

Action Plugins

This section applies to:

- AutoPlay Media Studio 5.0
- AutoPlay Media Studio 6.0
- Setup Factory 7.0
- TrueUpdate 2.0
- Visual Patch 2.0

Action plugins allow you to extend the actions that ship with the product. Using action plugins you can add new actions in a very seamless manner.

Required Skills

You will need the following skills to create action plugins:

- C or C++ programming
- Be able to create Windows DLLs
- A familiarity with the Lua C API
- Knowledge of working with XML

Action Plugin Files

Action plugin files are Windows DLLs that have the file extension ".lmd". These DLLs expose a certain set of exported functions that the product uses for informational and functional purposes.

The integration of action plugins at run time is done by mapping C functions into the run time's Lua engine. For more information about this process, please read the Lua Reference Manual, which is available from http://www.lua.org/manual. It is important that you have a good understanding of Lua's C API and how Lua works before creating action plugins.

Required Exported Functions

Below is a list and description of functions that must be exported from your action plugin DLL.

irPlg_GetPluginName

Purpose: To return the name of the action plugin to the calling program. Keep this short but descriptive. For example, if your plugin is used to access ADO databases, call it something like “ADODatabase”.

Prototype:

```
int irPlg_GetPluginName(char* szBuffer, int* pnBufferSize)
```

Parameters:

szBuffer - [out] A pointer to a character buffer that will receive the name of the plugin.
pnBufferSize - [in/out] A pointer to an integer that contains the number of characters in szBuffer on the way in and will be set to the number of characters actually copied to the buffer on the way out.

Returns:

The number of characters copied to the buffer or -1 if the buffer was not large enough to contain the plugin’s name. If you return -1, be sure that you set pnBufferSize to the number of characters actually required.

irPlg_GetPluginVersion

Purpose: To return the version of the plugin to the calling program. This version number is used to identify different versions of your plugin. It can be any version number, but it should be in the format “#.#.#.#”.

Prototype:

```
int irPlg_GetPluginVersion (char* szBuffer, int* pnBufferSize)
```

Parameters:

szBuffer - [out] A pointer to a character buffer that will receive the version of the plugin.
pnBufferSize - [in/out] A pointer to an integer that contains the number of characters in szBuffer on the way in and will be set to the number of characters actually copied to the buffer on the way out.

Returns:

The number of characters copied to the buffer or -1 if the buffer was not large enough to contain the plugin’s version. If you return -1, be sure that you set pnBufferSize to the number of characters actually required.

irPlg_ShowHelpForPlugin

Purpose: To show help information for the plugin. What exactly this function does is completely up to you. Usually opening an html document either locally or on the Internet is sufficient.

Prototype:

```
bool irPlg_ShowHelpForPlugin(char* lpszPluginPath, HWND hParentWnd)
```

Parameters:

lpszPluginPath - [in] A pointer to a character buffer that contains the folder that the plugin is currently located in. This can be useful if you want to open a help file from the same folder.
hParentWnd - [in] A handle to the window that is calling the function. It is sometimes necessary to have this value when opening files.

Returns: TRUE if the help was successfully displayed or FALSE if it failed.

irPlg_ShowHelpForAction

Purpose: To show help information for a specific action in the plugin. What exactly this function does is completely up to you. Usually opening an html document either locally or on the Internet is sufficient.

Prototype:

```
bool irPlg_ShowHelpForAction(char* lpszActionName, char* lpszPluginPath,
      HWND hParentWnd)
```

Parameters:

lpszActionName - [in] A pointer to a character buffer that contains the name of the action to show help for.
lpszPluginPath - [in] A pointer to a character buffer that contains the folder that the plugin is currently located in. This can be useful if you want to open a help file from the same folder.
hParentWnd - [in] A handle to the window that is calling the function. It is sometimes necessary to have this value when opening files.

Returns: TRUE if the help was successfully displayed or FALSE if it failed.

irPlg_GetAuthorInfo

Purpose: To return information about the plugin and its author. This information will be displayed in the About Plugin screen at design time.

Prototype:

```
int irPlg_GetAuthorInfo (char* szBuffer, int* pnBufferSize)
```

Parameters:

szBuffer - [out] A pointer to a character buffer that will receive the author info of the transition. This information can contain and be formatted in any way that you wish
pnBufferSize - [in/out] A pointer to an integer that contains the number of characters in szBuffer on the way in and will be set to the number of characters actually copied to the buffer on the way out.

Returns:

The number of characters copied to the buffer or -1 if the buffer was not large enough to contain the plugin's author information. If you return -1, be sure that you set pnBufferSize to the number of characters actually required.

irPlg_IsValidLicense

Purpose: To determine if license information for the plugin is valid. The license information comes from a license file that is located in the same folder as your plugin DLL at design time. License files are covered in more detail in the topic "License Files" on page 30.

Prototype:

```
bool irPlg_IsValidLicense (char* lpszLicenseInfo)
```

Parameters:

lpszLicenseInfo - [in] A pointer to a character buffer that contains text from the license file.

Returns:

TRUE is the license information is valid or FALSE if it is not. If you return FALSE, the user will not be able to use the transition.

irPlg_GetPluginActionXML

Purpose: The calling program will call this function to retrieve information about all of the actions that are provided in this plugin. The function should return the information in XML format. The XML formatting details are covered elsewhere in this document.

Prototype:

```
int irPlg_GetPluginActionXML(char* szBuffer, int* pnBufferSize)
```

Parameters:

szBuffer - [out] A pointer to a character buffer that will receive the XML.
pnBufferSize - [in/out] A pointer to an integer that contains the number of characters in szBuffer on the way in and will be set to the number of characters actually copied to the buffer on the way out.

Returns:

The number of characters copied to the buffer or -1 if the buffer was not large enough to contain the XML. If you return -1, be sure that you set pnBufferSize to the number of characters actually required.

irPlg_GetLuaVersion

Purpose: To tell the calling program which version of Lua the plugin links to.

Prototype:

```
int irPlg_GetLuaVersion(char* szBuffer, int* pnBufferSize)
```

Parameters:

szBuffer - [out] A pointer to a character buffer that will receive the Lua version.
pnBufferSize - [in/out] A pointer to an integer that contains the number of characters in szBuffer on the way in and will be set to the number of characters actually copied to the buffer on the way out.

Returns:

The number of characters copied to the buffer or -1 if the buffer was not large enough to contain the Lua version string. If you return -1, be sure that you set pnBufferSize to the number of characters actually required.

Note: This function should always return LUA_VERSION which is defined in lua.h.

irPlg_Action_RegisterActions

Purpose: To add the plugin's actions to the Lua engine at run time.

Prototype:

```
int irPlg_Action_RegisterActions(lua_State* L)
```

Parameters:

L - [in/out] A pointer to the lua_State structure that is used by the run time program. This instance of the Lua engine has been properly initialized so all that you need to do is to register your functions and variables.

Returns:

Zero if success, some other number if it fails.

irPlg_GetDependencies (OPTIONAL)

Purpose: To return additional dependency files required by the plugin. This function should return only the filenames of files needed by the plugin at run time in a bar ("|") separated string. The files must be located in the same folder as the plugin file at design time. For example, "support.dll|splash.png", tells the product to collect and bring along the files support.dll and splash.png with the plugin at build time. These files will be copied to the same folder as the plugin file. This interface is optional and does not need to be exposed if not applicable to the plugin.

Prototype:

```
int irPlg_GetDependencies (char* szBuffer, int* pnBufferSize)
```

Parameters:

szBuffer - [out] A pointer to a character buffer that will receive the list of dependency files.

pnBufferSize - [in/out] A pointer to an integer that contains the number of characters in szBuffer on the way in and will be set to the number of characters actually copied to the buffer on the way out.

Returns:

The number of characters copied to the buffer or -1 if the buffer was not large enough to contain the dependency files string. If you return -1, be sure that you set pnBufferSize to the number of characters actually required.

Specifying Action XML

The XML string that is returned by the irPlg_GetPluginActionXML function must be formatted in a specific format. This is the same format used by Indigo Rose to specify action information for the products' built-in actions. You can take a look at these files in the \Data\Actions subfolder of the product's application folder.

This XML is only for the use of the design environment. It has no real affect on how your actions are called or what they do. It is just there for the sake of the action wizard/editor and the intellisense editor when typing script. You do not have to provide this XML, but without it users may have a harder time using your plugin. However, if you are just making a plugin for your own use, you can just return an empty string as XML if you wish.

Here is some sample action XML data:

```
<ActionTemplates>
    <Action>
      <Name>IRClipboard.CopyText</Name>
      <Description>Copies text to the Windows clipboard.</Description>
      <ReturnValueType></ReturnValueType>
      <Arguments>
            <Arg>
                  <Name>Text</Name>
                  <Description>The text to copy to the
clipboard.</Description>
                  <Type>string</Type>
                  <Default></Default>
                  <Required>1</Required>
                  <EasyMode>
                        <Default>"My Text"</Default>
                        <DataType>string</DataType>
                        <Constraints>none</Constraints>
                  </EasyMode>
            </Arg>
      </Arguments>
    </Action>
    <Action>
      <Name>IRClipboard.GetText</Name>
      <Description>Retrieves text from the Windows
clipboard.</Description>
      <ReturnValueType>string</ReturnValueType>
      <Arguments>
```

```
      </Arguments>
    </Action>
    <Action>
      <Name>IRClipboard.IsTextAvailable</Name>
      <Description>Determines whether text is available on the Windows
clipboard.</Description>
      <ReturnValueType>boolean</ReturnValueType>
      <Arguments>
      </Arguments>
    </Action>
</ActionTemplates>
```

<ActionTemplates>

This is the main tag. This tag must surround all of the other action XML data.

<Action>

The tag that surrounds a single action. You can have one or more Action tags per file.

<Name>

The name of the action. Make sure that this name exactly matches the name that you mapped into the Lua engine. Also try to use the "dot" notation as it helps avoid naming conflicts. That is, use "MyPlugin.MyFunction" not just "MyFunction".

<Description>

The description of the action as it will appear in the action wizard at design time.

<ReturnValueType>

The return value type. Although Lua is a typeless language, this will indicate to the user what kind of return value to expect, if any. This field can be empty if your action does not return a value. Be sure to stick to "string", "number", "boolean" or "table" as return values.

Note: Even though Lua functions can return multiple return values, we do not recommend doing so in your action plugins because it breaks from the standard used in Indigo Rose products and will not work as well with the intellisense code editing and action wizard.

<Arguments>

This tag will surround all of your action's arguments. This can be an empty tag. Your action does not have to accept arguments.

<Arg>

Surrounds a single argument.

<Name>

The name of the argument. This argument name does not truly mean anything except to help describe the argument itself to the user.

<Description>

The argument's description. This should be a simple, one-line description of what the argument is for. The description will be seen by the user in the action wizard.

<Type>

The argument type. Although Lua is a typeless language, this will indicate to the user what kind of value to pass in. All arguments must have a type. Be sure to stick to "string", "number", "boolean", "table" or "variant" (meaning it can take any type of argument) as types.

<Default>

The default value of the argument. Use this only if your C code for the action is able to deal with an argument not being supplied. Also, all arguments that support defaults should be at the end of the arguments list together:

```
string MyPlugin.MyFunction(string Text, number Option = 1, boolean
Switch = true)
```

Note that this default value is only really significant when the user is typing script into the editor (when the tooltip with the function prototype appears), and means nothing to the action wizard.

<Required>

Whether the argument is required or optional. Use 0 for not required or 1 for required. As a rule, arguments without defaults are required and arguments with defaults are not required.

<EasyMode>

This section specifies information for the argument that will only be used in the action editor. The information will not be used when typing in script mode.

<Default>

The default value for the argument. This is not the same as the Default tag outside of the EasyMode section. This is simply the default value that will appear when the user creates a new action in the action wizard. It is there to help them out with a default value. You do not have to provide a default value.

<DataType>, <Constraints>

The type of data that the argument uses. This will help the action editor determine which kind of grid cell to use for the argument. The Constraints tag is used to further modify which kind of data that the grid cell will accept.

Here is a description of the various acceptable data types and the constraints that apply to them:

<DataType>	Description	<Constraints>
string	String data. This will accept any kind of textual input. The user can type in anything that they want (including a variable name or a function, etc.)	Can be "none" if the user can enter any amount of text. Otherwise, you can specify the number of characters in the format "#,#" Where # is any number and can also be * to indicate that there is no limit. Examples: 1,10 - Number of characters must be between 1 and 10 1,* - One or more characters 0,* - Any amount of characters. (same as using "none") Note that a user could also use a variable such as "x" here which would have only one character, but could contain any number of characters. For this reason, it is generally best to leave the constraints fairly open with strings unless you have a good reason not to.
number	Numerical data. This will accept any number or a variable.	Can be "none" if you don't want validation performed on the number. Otherwise, you can specify a contraint in the format "#,#" where # is any number that specifies a minimum or maximum acceptable value. Examples: 1,10 - Accepts a number between 1 and 10 (inclusive) 1,* - Accepts any positive number

		, - Accepts any number (same as using “none”) Note that a number field will also accept textual data because it could be a variable name (such as “MyNumber”). For this reason, data validation is only performed if the input is determined to be numeric. For example, the input “x1” would not be validated but “-98.23” would be.
boolean	A boolean value. This field will be a dropdown that contains “true” and “false” as well as accepting a variable value.	None.
combo	A dropdown combo will be presented with options that you want to offer the user. The user will also be able to type in a value or variable if they wish.	A comma separated list of combo options. Examples: “Apple”,”Orange”,”Pear” MY_CONST1,MY_CONST2 2,4,8,24,32
file	A field with a file selector button. The file that is selected will be brought into the project resource folder when the user selects it.	The type of files that you want to accept and browse for. The following values are acceptable: “Audio”, “Buttons”, “Docs”, “Flash”, “Images”, “Scripts” or “Videos” Note that using “Docs” will allow the user to browse for any type of file.
fileedit	Works exactly like “file” but allows the user to type into the field.	Same as the constraints for “file”
objectname	(*Supported in AutoPlay Media Studio only*) Shows a combo box filled with all of the object names of a certain type on the current page. The user can also type in a name or variable if they want to.	The type of object to display: "button", “label”, “paragraph”, “image”, “flash”, “video”, “web”, “input”, “hotspot”, “listbox”, “tree”, “combobox”, “progress”, “plugin” or “all”

		(tree, combobox and progress are only supported in AutoPlay Media Studio 6.0) The “plugin” constraint will show all plugins on the page regardless of the plugin type. Use the data type “pluginobject” if you want to filter by a certain type of plugin. The “all” option will display all objects on the page.
multiline	Multiline text editing. This will allow the user to type into the field as well as providing a browse button that will open the text editor complete with spell checking.	Same as the constraints for “string”.
pagename	(*Supported in AutoPlay Media Studio only*) Shows a combo box filled with the names of all pages currently in the project. The user can also type in a name or variable if they want to.	None.
proj_folder	An editable field with a browse button that allows the user to select a folder from their project.	None.
color	Shows a color selector field.	None.
pluginobject	(*Supported in AutoPlay Media Studio only*) Shows a combo box filled with plugin objects of a certain type.	The type of plugin object to accept. This will be the internal identifier of the plugin object which is usually only known by the plugin author.
Button	(*Supported in Setup Factory, TrueUpdate and Visual Patch only*) Shows a combo box filled with button objects.	None.
Button	(*Supported in Setup Factory, TrueUpdate and Visual Patch only*) Shows a combo box filled with Button objects.	None.
CheckBox	(*Supported in Setup Factory, TrueUpdate and Visual Patch only*) Shows a combo box filled with CheckBox objects.	None.
ComboBox	(*Supported in Setup Factory, TrueUpdate and Visual Patch*	None.

	only) Shows a combo box filled with ComboBox objects.	
EditField	(*Supported in Setup Factory, TrueUpdate and Visual Patch only*) Shows a combo box filled with EditField objects.	None.
ComboBox	(*Supported in Setup Factory, TrueUpdate and Visual Patch only*) Shows a combo box filled with ComboBoxobjects.	None.
ListBox	(*Supported in Setup Factory, TrueUpdate and Visual Patch only*) Shows a combo box filled with ListBox objects.	None.
ProgressBar	(*Supported in Setup Factory, TrueUpdate and Visual Patch only*) Shows a combo box filled with ProgressBar objects.	None.
RadioButton	(*Supported in Setup Factory, TrueUpdate and Visual Patch only*) Shows a combo box filled with RadioButton objects.	None.
ScrollingText	(*Supported in Setup Factory, TrueUpdate and Visual Patch only*) Shows a combo box filled with ScrollingText objects.	None.
SelectPackages	(*Supported in Setup Factory only*) Shows a combo box filled with SelectPackages objects.	None.
StaticText	(*Supported in Setup Factory, TrueUpdate and Visual Patch only*) Shows a combo box filled with StaticText objects.	None.

Including the Lua Library

The Lua library itself is the main required component that must be linked with your DLL. All of the Lua files are located in the \Includes subfolder of the SDK folder.

Required Files

lua.h

This is the main include file for the Lua library. It includes declarations for all of the base functionality of Lua.

lauxlib.h

This contains declarations for many useful functions that assist in the creation of Lua libraries (which is what the actions in our plugins are.)

lualib.h

Contains declarations for the standard Lua libraries such as string, math, etc.

lua.lib

The release version of the Lua library that will be statically linked into your plugins.

luaD.lib

The debug version of the Lua library which will be statically linked into your plugins. Make sure that you configure your C++ projects only to link with this library when doing debug builds. Do not use it for the release (final) versions of your plugins. Instead, use the "lua.lib" file.

Adding the Lua Library to Your Project

In order to link the Lua static library into your projects, there are just a few steps to follow:

1. Add lua.h, lualib.h, lauxlib.h, lua.lib and luaD.lib (optional) to your project.

In Visual C++, this can be done by selecting Project > Add to Project > Files from the menu. If you are not concerned with being able to debug the Lua library, you do not have to include luaD.lib.

2. Include three Lua header files in any other source files that need the Lua functions.

In C, simply include the following in your .c file that uses the Lua functions:

```
#include "lua.h"
#include "lauxlib.h"
#include "lualib.h"
```

In C++, use the following:

```
extern "C" {
#include "lua.h"
#include "lauxlib.h"
#include "lualib.h"
};
```

If you are using Visual C++ and are using pre-compiled headers, put the code above into your stdafx.h file and then it will be visible to the entire application.

3. Make sure that the lua*.lib files are linked into your build.

In Visual C++, this can be done by selecting Project > Settings from the menu and selecting the Link tab. Add luaD.lib to the "Object/library modules" field of your debug builds and lua.lib to the same field in your release builds.

More About the Lua Library

Although you can download and compile Lua for yourself from http://www.lua.org, we ask that you use the one provided by this SDK when making plugins. The reason for this is that statically linking to the library eliminates the need to distribute another DLL. As well, the Lua library that we provide has been slightly modified to use global heap memory allocation instead of per-process memory allocation, which can cause problems. Note that the product run times as well as the SDK Lua distribution links to the multithreaded static versions of the C run time libraries, so your plugins should do the same.

Including IRPluginHelperFunctions

Indigo Rose has provided a set of functions that may prove useful when developing your actions in plugin DLLs. These functions are located in the files IRPluginHelperFunctions.h and IRPluginHelperFunctions.cpp which are located in the \Includes subfolder of the SDK folder. The functions in these files are well commented within the source files and are used in many of the sample projects. Although you do not have to use these functions to make plugins, they can certainly help.

Distributing Action Plugins

The action plugin DLL (".lmd" file), the license file (".lic") and any accompanying help files should be copied to a unique subfolder of the \Plugins\Actions subfolder of the product installation folder in order to be visible to the design environment. For example, "\Plugins\Actions\MyPlugin".

Hints and Tips

Here are some things that may help you out when creating action plugins.

Sample Code

An entire sample Visual C++ 6.0 project is available for you to see and learn from. It is the source code used to produce the IRClipboard plugin that ships with all Indigo Rose products. This project is located in the \Samples\ IRClipboard subfolder of the SDK folder.

This project will show you all of the details behind making your own action plugin. Note that this DLL statically links to MFC for the sake of the properties dialog. However, it is possible to make action plugins that don't rely on MFC.

Test Your Plugin Thoroughly

Make sure to fully test your plugin before sharing it with others. It is important to test it thoroughly because a buggy plugin could cause the entire run time to crash.

Object Plugins

This section applies to:

- AutoPlay Media Studio 5.0
- AutoPlay Media Studio 6.0

Object plugins allow you to extend the AutoPlay Media Studio objects. Using object plugins you can add new objects in a very seamless manner.

Required Skills

You will need the following skills to create action plugins:

- C++ and object-oriented programming
- Be able to create Windows DLLs
- A familiarity with the Lua C API
- Knowledge of working with XML

Object Plugin Files

Object plugin files are Windows DLLs that have the file extension ".apo". These DLLs expose a certain set of exported functions that AutoPlay Media Studio uses for informational and functional purposes. These new objects can not only add visual elements to the product but can also add new actions as well.

The integration of an object plugins actions at run time is done by mapping C functions into the run time's Lua engine. For more information about this process, please read the Lua Reference Manual which is available from http://www.lua.org/manual. It is important that you have a good understanding of Lua's C API and how Lua works before creating action plugins.

Required Exported Functions

Below is a list and description of functions that must be exported from your object plugin DLL. Note that most of the functions are identical in form and function to those exposed by action plugins except where noted.

irPlg_GetPluginName

Please see the explanation for this function in the action plugin section on page 33.

irPlg_GetPluginVersion

Please see the explanation for this function in the action plugin section on page 33.

irPlg_GetPluginActionXML

Please see the explanation for this function in the action plugin section on page 35.

irPlg_ShowHelpForAction

Please see the explanation for this function in the action plugin section on page 34.

irPlg_ShowHelpForPlugin

Please see the explanation for this function in the action plugin section on page 33.

irPlg_IsValidLicense

Please see the explanation for this function in the action plugin section on page 35.

irPlg_GetLuaVersion

Please see the explanation for this function in the action plugin section on page 35.

irPlg_GetAuthorInfo

Please see the explanation for this function in the action plugin section on page 34.

irPlg_Object_CreateObject

Purpose: To create an instance of a CIRPluginObject-derived class and return a pointer to it. The CIRPluginObject class will be explained more in the next section.

Prototype:

```
CIRPluginObject* irPlg_Object_CreateObject()
```

Parameters:

None.

Returns:

A pointer to a CIRPluginObject-derived class.

irPlg_Object_DeleteObject

Purpose: Destroy an instance of a previously created CIRPluginObject-derived class. The CIRPluginObject class will be explained more in the next section.

Prototype:

```
void irPlg_Object_DeleteObject(CIRPluginObject* pObject)
```

Parameters:

pObject - [in] A pointer to a previously created CIRPluginObject-derived class.

Returns:

Nothing.

irPlg_GetIRPluginObjectVersion

Purpose: To return the version of the CIRPluginObject class to the calling program. This function should always return the defined constant IR_PLUGIN_CLASS_VERSION. This function is used to ensure future compatibility.

Prototype:

```
int irPlg_GetIRPluginObjectVersion ()
```

Parameters:

None.

Returns:

A numeric value which should always be the value IR_PLUGIN_CLASS_VERSION which is defined in "IRPluginObject.h".

irPlg_GetDependencies (OPTIONAL)

Please see the explanation for this function in the action plugin section on page 36.

irPlg_Object_GetFonts (OPTIONAL)

Purpose: To return information about fonts required by the plugin. This function will be called at build time and will cause any fonts used by the plugin to be collected by the internal font manager and included in the application. This interface is optional and does not need to be exposed if not applicable to the plugin.

Prototype:

```
int irPlg_Object_GetFonts (CIRPluginObject* pObject, char* szBuffer,
int* pnBufferSize)
```

Parameters:

pObject - [in] A pointer to a previously created CIRPluginObject-derived class.
szBuffer - [out] A pointer to a character buffer that will receive the font information. This information must be XML formatted in the following format:

```
<PluginFonts>
    <Font>
      <Weight></Weight>
      <Italic></Italic>
      <CharSet></CharSet>
      <FaceName></FaceName>
      <StyleName></StyleName>
    </Font>
</PluginFonts>
```

You may have one or more <Font> entries. These entries have the values from the Windows API LOGFONT structure. Usually your object will be using more information than the above to create and use fonts at runtime, but these are all that is needed to properly include the font at built time. Example:

```
 <PluginFonts>
    <Font>
      <Weight>700</Weight>
      <Italic>1</Italic>
      <CharSet>0</CharSet>
      <FaceName>Palatino Linotype</FaceName>
      <StyleName>Bold Italic</StyleName>
    </Font>
    <Font>
      <Weight>400</Weight>
      <Italic>0</Italic>
      <CharSet>0</CharSet>
      <FaceName>Arial</FaceName>
      <StyleName>Regular</StyleName>
    </Font>
</PluginFonts>
```

pnBufferSize - [in/out] A pointer to an integer that contains the number of characters in szBuffer on the way in and will be set to the number of characters actually copied to the buffer on the way out.

Returns:

The number of characters copied to the buffer or -1 if the buffer was not large enough to contain the font data. If you return -1, be sure that you set pnBufferSize to the number of characters actually required.

irPlg_Object_TranslateMessage (OPTIONAL)

Purpose: To pass window messages from the runtime engine to the object plugin. This is sometimes necessary if you are using a window with the style WS_POPUP from your plugin and are statically linking to the MFC libraries. In this case messages are not properly passed to the DLL CWinApp-derived message queue due to a limitation of the MFC libraries. You only need to implement this function if you are having troubles getting windows messages through to your windows. Most of the time you should not need to implement this function.

You can also use this function if you want to make a plugin that captures low-level window messages of the runtime engine itself.

Prototype:

```
BOOL irPlg_Object_TranslateMessage (MSG* pMsg)
```

Parameters:

pMsg - [in] A pointer to MSG structure (see MSDN for more details about this structure.)

Returns:

A boolean. Usually you should just return FALSE. The most common implementation is to pass the pMsg through to your application class' PreTranslateMessage function:

```
BOOL irPlg_Object_TranslateMessage(MSG* pMsg)
{
    return theApp.PreTranslateMessage(pMsg);
}
```

The CIRPluginObject Class

As you can see in the previous section, the functions irPlg_Object_CreateObject and irPlg_Object_DeleteObject are used to create and destroy a CIRPluginObject class. In this section we will look at this class closely and discover how it works.

Basically, the CIRPluginObject class is the basis of all plugin objects. All plugin objects must make a class which derives from this base class in order to be used by AutoPlay Media Studio. You can get the source code for this class from the \Includes subfolder of the SDK folder. The files are called IRPluginObject.cpp and IRPluginObject.h. Note that they are C++ files and can only be used as C++ classes. Unlike all other plugins presented so far, it is not possible to make object plugins without using C++.

So, in order to make an object plugin, first derive a class of your own from CIRPluginObject (derive publically). The rest is just a matter of filling in the virtual functions and adding your own member variables and functions to make things happen. Below is a list of the member variables and functions exposed by CIRPluginObject.

CIRPluginObject::GetDefaultSize

Purpose: To return the default size that a new object of this type should be.

Prototype:

```
virtual void GetDefaultSize(SIZE* pSize)
```

Parameters:

pSize - [in] A pointer to a SIZE structure for your class to fill in.

Returns:

Nothing.

CIRPluginObject::IsWindowedObject

Purpose: To return whether or not the plugin is a windowed object. Windowed objects are objects that will create their own window for the object. A non-windowed object is one that will draw itself directly onto the run time window's client area. For example, the Shape object that ships with AutoPlay Media Studio is a non-windowed object whereas the Slider one is. The only real difference that this makes to the run time engine is how it passes screen co-ordinates to the plugin.

Prototype:

```
virtual BOOL IsWindowedObject()
```

Parameters:

None.

Returns:

TRUE if the plugin is a windowed object or FALSE if not.

CIRPluginObject:: GetWindowHandle

Purpose: To return the handle to the object's window, if it is a windowed object. The run time uses this handle for things such as determining if the object has input focus.

Prototype:

```
virtual HWND GetWindowHandle()
```

Parameters:

None.

Returns:

The handle to the object's window, if it has one. Otherwise NULL.

CIRPluginObject::DrawDesign

Purpose: This function draws the object on the screen at design time. This function will only be called when the object is displayed at design time.

Prototype:

```
virtual void DrawDesign(HDC hDC, HWND hMainWnd, RECT rcObRect,
          BOOL bVisible, BOOL bEnabled)
```

Parameters:

hDC - [in] A handle to the device context that the object can use to draw to.
hMainWnd - [in] A handle to the parent window of the object (the design environment's page view area.)
rcObRect – [in] A RECT structure that contains the coordinates and size of the object relative to the upper-left corner of the page area. The object should not draw outside of this area.
bVisible – [in] Whether the object is visible at design time or not. If FALSE, the function should not draw the object and should destroy or hide the object's window if applicable.
bEnabled – [in] Whether the object should be drawn in a disabled state. Objects do not have to show a disabled representation.

Returns:

Nothing.

CIRPluginObject::DrawRun time

Purpose: This function draws the object on the screen at run time. This function will only be called when the object is displayed at run time.

Prototype:

```
virtual void DrawRun time(HDC hDC, HWND hMainWnd, RECT rcObRect,
          BOOL bVisible, BOOL bEnabled)
```

Parameters:

hDC - [in] A handle to the device context that the object can use to draw to.
hMainWnd - [in] A handle to the parent window of the object (the run time window's client area)
rcObRect – [in] A RECT structure that contains the coordinates and size of the object. The object should not draw outside of this area.
bVisible – [in] Whether the object is visible at run time or not. If FALSE, the function should not draw the object and should destroy or hide the object's window if applicable.
bEnabled – [in] Whether the object should be drawn in a disabled state. Objects do not have to show a disabled representation.

Returns:

Nothing.

CIRPluginObject::GetCustomProperties

Purpose: Gets the custom properties of an object plugin in its current state. This function can return any type of string data in any format. The contents of the string is up to the plugin to interpret.

Prototype:

```
virtual int GetCustomProperties(char* szBuffer, int* pnBufferSize)
```

Parameters:

szBuffer - [out] A pointer to a character buffer that will receive the custom properties of the object.
pnBufferSize - [in/out] A pointer to an integer that contains the number of characters in szBuffer on the way in and will be set to the number of characters actually copied to the buffer on the way out.

Returns:

The number of characters copied to the buffer or -1 if the buffer was not large enough to contain the properties. If you return -1, be sure that you set pnBufferSize to the number of characters actually required.

CIRPluginObject::SetCustomProperties

Purpose: Sets the custom properties of the object. This information was previously obtained using the GetCustomProperties function. It is up to the plugin to interpret this data in a way that makes sense to it.

Prototype:

```
virtual void SetCustomProperties(char* szPropsList)
```

Parameters:

szPropsList - [in] A pointer to a string that contains the plugin's properties.

Returns:

Nothing.

CIRPluginObject::ShowProperties

Purpose: Shows the properties dialog (if any) of the plugin. Thiswill be called when the user tries to edit the plugin object's custom properties at design time. How this dialog looks and operates is completely up to you, the plugin developer.

Prototype:

```
virtual BOOL ShowProperties(char* szPluginFolder)
```

Parameters:

szPluginFolder - [in] A pointer to a string that contains the location of the plugin file (.apo). This can be useful if you want to provide a Help button on your properties dialog and may need to know the location of a help file.

Returns:

TRUE if the user made changes to the object through the properties dialog or FALSE if they did not (or if they did and then cancelled.)

CIRPluginObject::GetNumEvents

Purpose: To return the number of events supported by the object. These are the events that will be exposed to the AutoPlay Media Studio developer at design time and fired at run time.

Prototype:

```
virtual int GetNumEvents()
```

Parameters:

None.

Returns:

The number of events supported by the object. Return 0 if the object does not support events.

CIRPluginObject::GetEvent

Purpose: To fill an IRPluginEventInfo structure with information about an event.

Prototype:

```
virtual BOOL GetEvent(int nIndex, IRPluginEventInfo* pEventInfo)
```

Parameters:

nIndex - [in] The index of the event to return information about.
pEventInfo – [out] A pointer to an already allocated IRPluginEventInfo structure. This structur is defined in IRPluginObject.h. It consists of the name of the event as well as a prototype for any event arguments that it supports.

Returns:

TRUE if the requested event index was found and the structure successfully filled with event information.

CIRPluginObject::RegisterLUAFunctions

Purpose: To register any actions that might be provided by the plugin with the Lua engine at run time.

Note: Your derived class should ALWAYS call the base class function before making its own modifications to the Lua engine: `CIRPluginObject::RegisterLUAFunctions(L);` Doing this will ensure that the class' m_pLuaState member variable gets set properly.

Prototype:

```
virtual int RegisterLUAFunctions(lua_State* L)
```

Parameters:

L - [in/out] A pointer to the lua_State structure that is used by the run time program. This instance of the Lua engine has been properly initialized so all that you need to do is to register your functions and variables.

Returns:

0 if successful, any other number if not.

CIRPluginObject::LetAMSHandleCursorChange

Purpose: To tell the run time engine whether you want it to handle cursor changes according to the Plugin object's properties at design time or not. In general, this is only really an option for non-windowed objects. It is unlikely that AutoPlay Media Studio run time will be able to automatically change the cursor for windowed objects, so they should generally return FALSE.

Prototype:

```
virtual BOOL LetAMSHandleCursorChange()
```

Parameters:

None.

Returns:

TRUE if the AutoPlay Media Studio run time should handle cursor changes automatically or FALSE if the plugin handles it internally.

CIRPluginObject::LetAMSHandleSounds

Purpose: To tell the run time engine whether you want it to handle mouse over and mouse down sounds according to the Plugin object's properties at design time or not. In general, this is only really an option for non-windowed objects. It is unlikely that AutoPlay Media Studio run time will be able to automatically play sounds for windowed objects, so they should generally return FALSE.

Prototype:

```
virtual BOOL LetAMSHandleSounds()
```

Parameters:

None.

Returns:

TRUE if the AutoPlay Media Studio run time should handle sounds automatically or FALSE if the plugin handles it internally.

CIRPluginObject::LetAMSHandleTooltip

Purpose: To tell the run time engine whether you want it to handle the displaying of tooltips according to the Plugin object's properties at design time or not. In general, this is only really an option for non-windowed objects. It is unlikely that AutoPlay Media Studio run time will be able to automatically display tooltips for windowed objects, so they should generally return FALSE.

Prototype:

```
virtual BOOL LetAMSHandleTooltip()
```

Parameters:

None.

Returns:

TRUE if the AutoPlay Media Studio run time should handle tooltip displaying automatically or FALSE if the plugin handles it internally.

CIRPluginObject::CanSetFocus

Purpose: To tell the run time engine whether your object is capable of having input focus. In general only windowed objects can have input focus.

Prototype:

```
virtual BOOL CanSetFocus()
```

Parameters:

None.

Returns:

TRUE if the plugin's object can have input focus or FALSE if not.

CIRPluginObject::DoSetFocus

Purpose: To set the input focus to the object. In general, only windowed objects can have the input focus.

Prototype:

```
virtual void DoSetFocus()
```

Parameters:

None.

Returns:

Nothing.

CIRPluginObject::OnMouseOver

Purpose: To handle the mouse moving over the object's area at run time. Note that windowed objects may not ever get this message from the run time but will have to catch this in their own message handlers.

Prototype:

```
virtual void OnMouseOver(HWND hWndParent, POINT ptMousePos,
                  RECT rcObRect)
```

Parameters:

hWndParent – [in] A handle to the run time window's page view area.
ptMousePos – [in] A POINT structure containing the mouse coordinates relative to the upper-left corner of the run time window's client area.
rcObRect – [in] A RECT structure that contains the object's coordinates and size relative to the upper-left corner of the page area.

Returns:

Nothing.

CIRPluginObject::OnMouseLeave

Purpose: To handle the mouse leaving the object's area at run time. Note that windowed objects may not ever get this message from the run time but will have to catch this in their own message handlers.

Prototype:

```
virtual void OnMouseLeave(HWND hWndParent, POINT ptMousePos,
                  RECT rcObRect)
```

Parameters:

hWndParent – [in] A handle to the run time window's page view area.
ptMousePos – [in] A POINT structure containing the mouse coordinates relative to the upper-left corner of the run time window's client area.
rcObRect – [in] A RECT structure that contains the object's coordinates and size relative to the upper-left corner of the page area.

Returns:

None.

CIRPluginObject::OnLBtnDown

Purpose: To handle the left mouse button being pressed down object's area at run time. Note that windowed objects may not ever get this message from the run time but will have to catch this in their own message handlers.

Prototype:

```
virtual void OnLBtnDown(HWND hWndParent, POINT ptMousePos,
                  RECT rcObRect)
```

Parameters:

hWndParent – [in] A handle to the run time window's page view area.
ptMousePos – [in] A POINT structure containing the mouse coordinates relative to the upper-left corner of the run time window's client area.
rcObRect – [in] A RECT structure that contains the object's coordinates and size relative to the upper-left corner of the page area.

Returns:

Nothing.

CIRPluginObject::OnLBtnUp

Purpose: To handle the left mouse button being released over object's area at run time. Note that windowed objects may not ever get this message from the run time but will

have to catch this in their own message handlers. This is generally where an "On Click" event would take place.

Prototype:

```
virtual void OnLBtnUp(HWND hWndParent, POINT ptMousePos,
                  RECT rcObRect)
```

Parameters:

hWndParent – [in] A handle to the run time window's page view area.
ptMousePos – [in] A POINT structure containing the mouse coordinates relative to the upper-left corner of the run time window's client area.
rcObRect – [in] A RECT structure that contains the object's coordinates and size relative to the upper-left corner of the page area.

Returns:

Nothing.

CIRPluginObject::OnLBtnDoubleClick

Purpose: To handle the left mouse button being double-clicked over object's area at run time. Note that windowed objects may not ever get this message from the run time but will have to catch this in their own message handlers.

Prototype:

```
virtual void OnLBtnDoubleClick(HWND hWndParent, POINT ptMousePos,
                  RECT rcObRect)
```

Parameters:

hWndParent – [in] A handle to the run time window's page view area.
ptMousePos – [in] A POINT structure containing the mouse coordinates relative to the upper-left corner of the run time window's client area.
rcObRect – [in] A RECT structure that contains the object's coordinates and size relative to the upper-left corner of the page area.

Returns:

Nothing.

CIRPluginObject::OnRBtnDown

Purpose: To handle the right mouse button being pressed down object's area at run time. Note that windowed objects may not ever get this message from the run time but will have to catch this in their own message handlers.

Prototype:

```
virtual void OnRBtnDown(HWND hWndParent, POINT ptMousePos,
                  RECT rcObRect)
```

Parameters:

hWndParent – [in] A handle to the run time window's page view area.
ptMousePos – [in] A POINT structure containing the mouse coordinates relative to the upper-left corner of the run time window's client area.
rcObRect – [in] A RECT structure that contains the object's coordinates and size relative to the upper-left corner of the page area.

Returns:

Nothing.

CIRPluginObject::OnRBtnUp

Purpose: To handle the right mouse button being released over object's area at run time. Note that windowed objects may not ever get this message from the run time but will have to catch this in their own message handlers.

Prototype:

```
virtual void OnRBtnUp(HWND hWndParent, POINT ptMousePos,
                  RECT rcObRect)
```

Parameters:

hWndParent – [in] A handle to the run time window's page view area.
ptMousePos – [in] A POINT structure containing the mouse coordinates relative to the upper-left corner of the run time window's client area.
rcObRect – [in] A RECT structure that contains the object's coordinates and size relative to the upper-left corner of the page area.

Returns:

Nothing.

CIRPluginObject::OnRBtnDoubleClick

Purpose: To handle the right mouse button being double-clicked over object's area at run time. Note that windowed objects may not ever get this message from the run time but will have to catch this in their own message handlers.

Prototype:

```
virtual void OnRBtnDoubleClick(HWND hWndParent, POINT ptMousePos,
                  RECT rcObRect)
```

Parameters:

hWndParent – [in] A handle to the run time window's page view area.
ptMousePos – [in] A POINT structure containing the mouse coordinates relative to the upper-left corner of the run time window's client area.
rcObRect – [in] A RECT structure that contains the object's coordinates and size relative to the upper-left corner of the page area.

Returns:

Nothing.

CIRPluginObject::FireEvent

Purpose: Allows the object to fire one of its own events. This is not a virtual function. It is part of the base class to provide this functionality.

Prototype:

```
void FireEvent(LPCTSTR strEventName, LPCTSTR strArguments)
```

Parameters:

strEventName – [in] The name of the event to fire.
strArguments – [in] A string containing the arguments to be passed to the event. Note that the arguments should be in the form of properly formed Lua script as it will be passed to the event verbatim.

Returns:

Nothing.

CIRPluginObject::GetObjectID

Purpose: Gets the unique ID of the plugin object. This is not a virtual function so you should not override it. All that this function basically does is to return the value of the member variable m_szObjectID. Make sure to set this value in your constructor. This ID must be completely unique to your plugin.

Prototype:

```
int GetObjectID(char* szBuffer, int* pnBufferSize)
```

Parameters:

szBuffer - [out] A pointer to a character buffer that will receive the ID of the object.
pnBufferSize - [in/out] A pointer to an integer that contains the number of characters in szBuffer on the way in and will be set to the number of characters actually copied to the buffer on the way out.

Returns:

The number of characters copied to the buffer or -1 if the buffer was not large enough to contain the properties. If you return -1, be sure that you set pnBufferSize to the number of characters actually required.

CIRPluginObject::ShowWindow

Purpose: Called at run time to have the plugin object show or hide its window. It is usually only necessary to implement this function in your derived class if your plugin is a windowed object.

Prototype:

```
void ShowWindow(BOOL bVisible)
```

Parameters:

bVisible - [in] A boolean that tells the function whether to show the window (TRUE) or hide it (FALSE).

Returns:

Nothing.

CIRPluginObject::m_pLuaState

This member variable holds a pointer to the run time engine's lua_State structure. This way it can be accessed from functions throughout the class.

CIRPluginObject::m_szObjectID

This member variable is a character array that holds the unique identifier of the plugin. This identifier is not seen by the user but is used for internal purposes. Make sure to set this variable in your derived class' constructor.

IRLUA_PLUGIN_GetObjectPtr

Purpose: Although this function is not part of the class itself, it is a helper function that is defined in IRPluginObject.h. It is used to pass in the name of a plugin object and to retrieve a pointer to the CIRPluginObject that implements it. This function is really useful when writing actions that operate on the plugin itself.

Prototype:

```
CIRPluginObject* IRLUA_PLUGIN_GetObjectPtr(lua_State *luaState,
                        LPCTSTR strObjectName)
```

Parameters:

luaState - [in] A pointer to the run time engine's lua_State structure.
strObjectName - [in] The name of the object that you want to get the pointer to. This is the name of the object as assigned to it by the developer at design time.

Returns:

A pointer to the CIRPluginObject that is associated with the named object. You can then cast this to your derived class and use it as you wish.

IRLUA_PLUGIN_RedrawObject

Purpose: Although this function is not part of the class itself, it is a helper function that is defined in IRPluginObject.h. It is used to force the run time engine to redraw the object that you name. This is useful when you have made changes to a property of the plugin through actions and you need to force the display to update the object.

Prototype:

```
void IRLUA_PLUGIN_RedrawObject(lua_State *luaState,
            LPCTSTR strObjectName)
```

Parameters:

luaState - [in] A pointer to the run time engine's lua_State structure.
strObjectName - [in] The name of the object that you want to redraw.

Returns:

Nothing.

Distributing Object Plugins

The object plugin DLL (".apo" file), the license file (".lic") and any accompanying help files should be copied to a unique subfolder of the \Plugins\Objects subfolder of the AutoPlay Media Studio installation folder in order to be visible to the design environment. For example, "\Plugins\Objects\MyPlugin".

Hints and Tips

Here are some things that may help you out when creating object plugins.

Sample Code

Two sample Visual C++ 6.0 projects are available for you to see and learn from. They are the source code used to produce the Shape and Slider plugins that ship with AutoPlay Media Studio. The projects are in the \Samples\Slider and Samples\Shape subfolders of the SDK folder.

These projects will show you how to make both windowed and non-windowed objects. Note that this DLL statically links to MFC for the sake of the properties dialogs. However, it is possible to make object plugins that don't rely on MFC.

Test Your Plugin Thoroughly

Make sure to fully test your plugin before sharing it with others. It is important to test it thoroughly because a buggy plugin could cause the entire run time to crash.